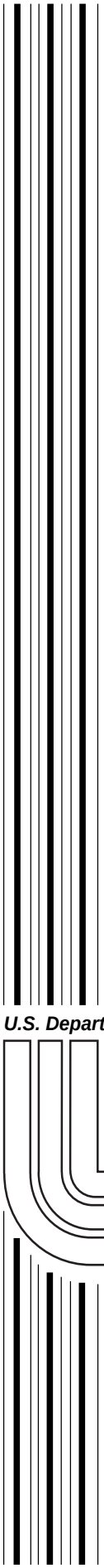

UCRL-ID-151790

Exploration of SMP-Aware DAO Memory Performance Issues—Final Report 2002

Bronis R. de Supinski, Andy Yoo, Sally A. McKee, Martin Schulz, and Tushar Mohan

February 4, 2003

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

Exploration of SMP-Aware DAO Memory Performance Issues
Final Report 2002

Bronis R. de Supinski, Andy Yoo

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
{bronis,ayoo}@llnl.gov

Sally A. McKee*and Martin Schulz

School of Electrical and Computer Engineering
Cornell University
Ithaca, NY 14853

{sam,schulz}@csl.cornell.edu

Tushar Mohan

School of Computing
University of Utah
Salt Lake City, UT 84112

tushar@cs.utah.edu

February 6, 2003

1 Introduction

The performance of many LLNL applications is dominated by the cost of main memory accesses. Worse, many current trends in computer architecture will lead to substantial degradation of the percentage of peak performance obtained by these codes. This project yields novel techniques that alleviate this problem in SMP-based systems, which are common at LLNL. Further, our techniques will complement other emerging mechanisms for improving memory system performance, such as processor-in-memory. The exploration of existing dynamic access ordering (DAO) mechanisms adapted to SMPs and the development of new memory performance optimization techniques will lead to significant improvements in run times for LLNL applications on future computing platforms, effectively increasing the size of the platform.

*This work was performed while Dr. McKee was at the Unive rsity of Utah; she has since relocated to Cornell University.

In this project, we have focused on a range of techniques to overcome the performance bottleneck of current multiprocessor systems and to increase the single-node efficiency. These efforts include the design and implementation of a toolset to analyze memory access patterns of applications, the exploration of regularity metrics and their use to classify code behavior, and a set of microbenchmarks to assess and quantify the performance of SMP memory systems. We will make these tools available to the general laboratory user community to help the evaluation and optimization of LLNL applications.

In addition, we explored the use of Dynamic Access Ordering (DAO) techniques in the realm of shared memory multiprocessors. The most critical part of the latter is the need to maintain coherence among reordered accesses due to possible aliasing. We have worked on several design alternatives to guarantee consistency in such systems without changing the user environment. This guarantees that such novel memory systems will be directly applicable for existing and future HPC codes at LLNL.

In the following section we will detail these efforts and discuss the results achieved within this project

2 Memory Behavior Studies

In the first part of this project we focused on memory access regularity of applications. We defined metrics to quantify regularity in codes, and developed a toolset to evaluate existing codes and to locate potentials for optimizations. This work is summarized in Tushar Mohan's University of Utah Master's of Science thesis [**?**], and was the subject of a seminar presented at LLNL on December 3, 2002. Our work has demonstrated that applications with highly regular access patterns are amenable to traditional optimization techniques, although they can also benefit from DAO mechanisms. Applications with irregular access patterns require more advanced methods, such as DAO mechanisms.

2.1 Motivation

Over the last two decades the concept of locality has been well understood and exploited. Spatial locality is the likelihood that a location *near* a recently accessed location will be accessed. Temporal locality refers to the likelihood that a recently accessed location will be accessed *again*. The success of and the current dependence on caching are testimony to the presence of locality in common codes. Locality concepts are limited to references with proximate or repeating accesses, and cannot capture the existence of other patterns. On the other hand, modern processor architectures and memory controllers are capable of exploiting the presence of other kinds of access patterns, such as strided memory accesses (successive memory accesses that have a constant difference in address). The Power3 processor, for instance, can detect and prefetch such strided accesses [3]. The Impulse memory controller can be configured to prefetch and gather strided memory elements [2]. To better understand (so that we may improve) the performance of codes on such architectures, we find it useful to extend the concept of locality to include the presence of strided memory accesses.

2.2 Regularity Metrics

Regular sequences, or streams, are precisely arithmetic progressions, defined as:

$$x_n = x_{n-1} + c$$

where c is a constant and x_n is the n^{th} reference in the regular sequence. As an illustration, consider the following series of numbers: `0 4 8 12 16 20`. This constitutes a regular sequence with stride four, length six and starting element `0`. The following metric can be used to quantify the spatial regularity of an application:

$$R_{spatial} = \frac{\sum \mid s_i \mid}{N}$$

where $\mid s_i \mid$ is the length of the i^{th} sequence and N is the total number of references. If we do not allow a reference to be included in more than one sequence, the metric is a positive number not greater than unity. Higher metric values imply greater spatial regularity.

We have used this regularity metric to separate applications into two classes: regular and irregular. We have used this distinction and other stream statistics, such as the mean length of sequences and variance in sequence lengths, to optimize codes. More details can be found in [**?**].

2.3 Toolset to Quantify Regularity in Applications

Based on these metrics, we have developed a toolset to detect and to quantify memory access behavior in applications. This toolset applies the metrics described above, and hence aids in the characterization of applications and their access patterns. It works directly on program executables (binaries), rather than on source code. This guarantees the broadest possible applicability. To achieve this goal, we use existing tools for dynamic instrumentation.

Figure 1 shows the setup for the stream detection tool. It consists of three separate components: `dsd, mutator` and the binary to be instrumented. The `mutator` is a generic application (written in C++) that uses the *Dyninst* library [1] to instrument the binary of interest. The instrumentation can be one or both of:

- a call for hardware performance monitoring at function entry/exits;
- a call to handle load/store information at every memory access.

A single function, `__instr__`, handles both calling contexts, with the arguments determining the context. The `mutator` can be configured to instrument specific functions and modules rather than the whole binary. Once the binary has been successfully instrumented, the `mutator` starts `dsd` – the stream detection module. At this time, the `mutator` passes control information, such as its process ID, and the functions instrumented and their *ids*, to `dsd` using the control channel (shown by the bi-directional arrow between the `mutator` and `dsd`). The `mutator` then starts the instrumented binary. The first time `__instr__` is called, it sets up one-way data and control pipes between itself (the instrumented application) and `dsd`. Conceptually, the control pipes carry information about function entry/exits, while the data pipes are used to pass PAPI counter values and

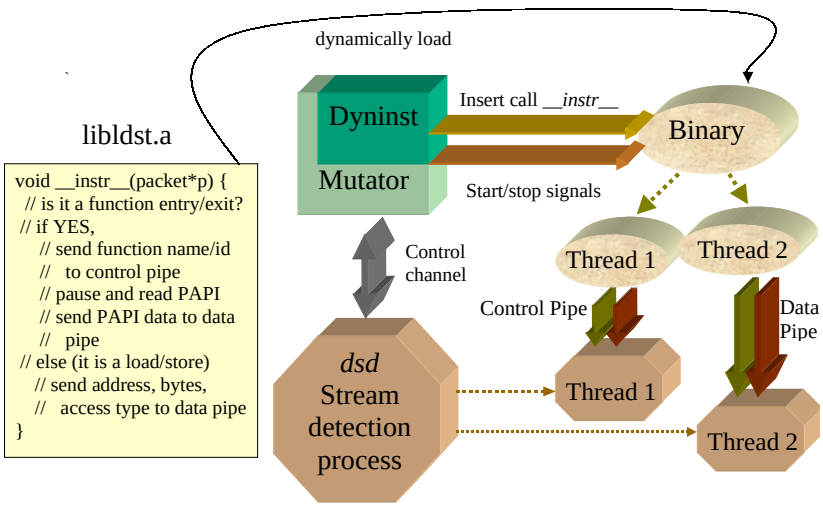

Figure 1: Stream Detection Framework

memory access information.[1] The direction of information flow is *from* the application *to* `dsd`. Subsequent calls to `__instr__` pass hardware performance counter values or memory access information (type, address and number of bytes) to `dsd`. The kernel automatically blocks the application when the data pipe fills up — typically, `dsd` processes data slower than it is generated. In the event that `dsd` wants to stop sampling a particular function (perhaps because it is configured to accept a limited number of samples), it signals the `mutator` using the control channel. The `mutator` pauses the application, deletes the inserted snippets from the function, and resumes the application.

2.4 Results

We have applied our tool to three real applications — `gzip`,`umt98` and `smg2000`[2] — and numerous benchmarks in standard use for architectural and performance explorations. Using our tool we:

- found and implemented new optimizations in these codes; and
- suggested optimizations that are consistent with earlier work on these codes.

Our results are promising: for `gzip` we uncovered and implemented two optimizations that reduce overall memory stalls by a few percent; in FT (a Fast Fourier Transform benchmark) we implement a loop interchange that reduced overall TLB misses and memory stalls by 58% and 8%, respectively. In many cases our tool suggests optimizations previously known to benefit the code, while in other cases we suggest new ones. In most of the codes analyzed, the tool indicates the function(s) to optimize and a set of applicable optimizations, significantly simplifying the optimization process. Table 1

[1] Actually, we multiplex the two information flows over a single channel for efficiency and synchronization reasons.

[2] `umt98` and `smg2000` are production codes in use at LLNL.

lists the regularity statistics for the codes. A detailed analysis for each application can be found in [?]. With the exception of `mgrid` (which is unoptimized), the data in the table is for an execution of the optimized binary (`-O2` or higher).

Program	Reg.	Streams					Stream Length		Stride	Optimization
		Total	]4-32]	]32-128]	]128-16384]	> 16384	Mean	Dev.		
gzip	0.95	5402	4625	76	17	584	3552	170	1.71	aggressive prefetching, super-paging and code restructuring
umt98	0.44	310655	307386	1818	1377	44	31	570	9.4	scatter/gather using IV in snswp3d
smg2000	0.36	100675	100519	78	74	4	13	108	6.95	code restructuring
mgrid	0.99	82359	52	82307	0	0	91	3.2	8.0	prefetching and tiling in RESID
swim	1.00	38259	1052	2	37188	17	614	223	4.4	aggressive prefetching, blocking, padding
su2cor	1.00	50274	33688	289	16317	0	1208	52	10.0	code restructuring, fission, padding, tiling, prefetching
CG	0.58	184954	143587	40555	805	7	46	518	5.4	scatter/gather using IV
FT	0.96	755750	734755	20481	513	1	12	23	109.3	loop interchange, array transposition
BT	0.80	1470357	1419110	51247	0	0	9	4.8	64.8	copying, base-stride remapping
Untiled 3D_JACOBI	1.00	77127	99	76832	196	0	85	26	12.0	tiling
Tiled 3D_JACOBI	1.00	61741	61741	0	0	0	11	1	8	none
Untiled matmult	1.00	88683	8081	0	80602	0	183	4.2	806.0	tiling
Tiled matmult	0.97	1788850	1788805	0	45	0	10	0.5	356.3	none

Table 1: Regularity Statistics

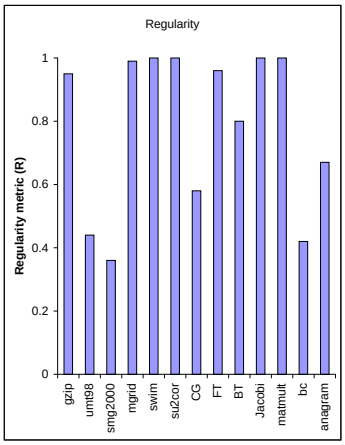

Figure 2: Regularity Metric

Figure 2 shows the regularity metric for all the applications. Notice the high regularity in stencil codes such as `mgrid, swim` and `3D_Jacobi`, and the far lower metric values for indirection-vector codes like `umt98` and `CG`. The results are for differing sampling modes: some sample the complete run of the application, while others take a few samples of important functions.

In general, high stride values can imply the need for loop interchange and copying/remapping optimizations for regular codes. We implement a loop interchange in `FT` and suggest copying/remapping for `BT`. Tiling the `matmult` and 3D_Jacobi kernels reduces their mean stream strides. `gzip`'s low mean stride is the result of a byte-by-byte scan for repeated strings.

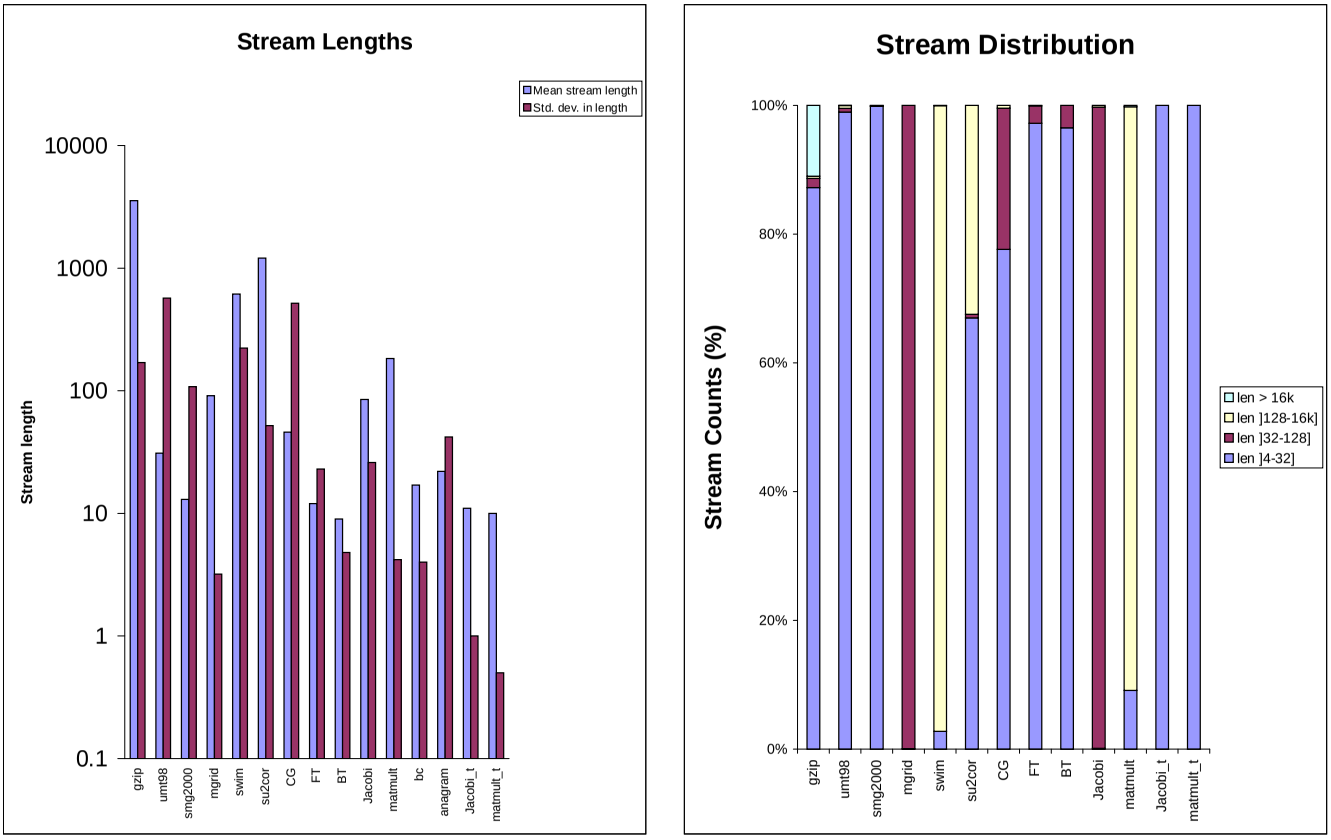

Figure 3: Stream Lengths

Mean stream lengths and the distribution of streams with lengths is shown in Figure 3. Stream lengths are often indicative of applicable optimizations. Applications with many long streams benefit from tiling; `mgrid, swim, su2cor, 3D_Jacobi` and `matmult` fall in this category. Tiled codes exhibit far lower mean stream lengths and variances; we see this when we compare untiled `matmult` and `3D_Jacobi` with their tiled counterparts. A high variance in stream lengths is present in many irregular codes, such as `umt98` and `CG`. Regular codes with long streams benefit from prefetching (sequential or stream), and this is verified for `gzip, swim` and `mgrid`.

3 Memory Microbenchmarks

In symmetric multiprocessors (SMPs), which have become the dominant components of high performance computer (HPC) systems in use at LLNL, several CPUs can simultaneously access a shared main memory. Applications running on an HPC are usually parallelized to take advantage of the SMP's concurrent memory access capability. How well the memory system supports the concurrent accesses varies widely — for instance, bus-based systems offer little support for concurrent accesses to main memory, while switch-based systems claim to eliminate this problem. Unfortunately, no existing memory benchmarks previously specifically addressed measuring the effect of concurrent accesses on SMP memory system performance.

We developed a suite of microbenchmarks to measure the memory performance of SMP-based machines. These benchmark are automatically generated (from a set of input options) to cover a large range of different memory access patterns. To test the effect of concurrent memory accesses on memory performance, each benchmark is multithreaded and exposes the memory system under investigation to a varying number of concurrent streams. This allows us to measure the interference of individual streams as well as to detect bottlenecks and critical loads for different memory architectures.

Portability is an important design goal, and thus we have implemented the suite platform independent and using only standard libraries commonly available on many systems. To the best of our knowledge, these are be the first benchmarks specifically targeted for properly measuring the memory performance of SMP-based machines running parallel applications.

We have used our benchmarks to measure the memory performance of *Blue-Pacific* (or *Blue*) and *Snow*, two SMP-based, high performance parallel computers in operation for the Accelerated Strategic Computing Initiative (ASCI) program at LLNL. Our results demonstrate that thread-level parallelism significantly impacts memory performance even for a memory interconnection designed to support concurrent memory accesses. Furthermore, we find that that concurrent accesses significantly reduce the benefits of hardware prefetching.

4 SMP-Aware DAO Techniques

The approaches discussed so far can only affect the memory behavior within the boundaries of existing memory systems. To study further improvements, it is necessary to consider modifications within the memory system hardware itself. One very promising approach in this direction is the use of remapping techniques, as done, e.g., in the Impulse system. Data items that are being accessed together but are distributed over

several cache lines are mapped into a single cache line and can be transferred into the L1 cache in one cache-line fill operation. Similarly, multiple pages can be combined to super pages, which reduces pressure on the TLB[3]. Both techniques can drastically reduce the memory traffic within the whole system as data is packed more densely and bus capacities are used more effectively. The core component for such a system is a modified memory controller that implements address remapping within the main memory of the system. It is programmed from within the applications, either explicitly by the programmer or by using an appropriate compiler/runtime infrastructure.

4.1 Alias Conflicts under Remapping

Remapping data, however, leads to scenarios in which the same physical data is present within the system at several virtual locations concurrently. E.g., an item that has been remapped can be accessed both through the remapping and at the same time at its original location. A consequence of this observation is that multiple copies of the same physical data can be present in the system caches under different mappings (or "names"), leading to potential inconsistencies between those copies. This disrupts the standard programming model, which is based on a fully consistent memory.

These coherence issues are easier to handle in uniprocessor and single-thread executions, as all accesses happen within a single thread of execution and their relative order is predictable and controllable. In multiprocessor systems, several processors concurrently access data with potentially different mappings. In order to successfully deploy such remapping schemes in a multiprocessor scenario, the system has to be augmented to transparently prevent such inconsistencies, and thereby to provide the expected programming model to the end user.

4.2 Novel Coherence Controller

The work within this project includes the design and evaluation of hardware extensions to existing cache coherence schemes for shared memory multiprocessor systems. They track existing copies of physical and remapped data and ensure proper updates or invalidations whenever necessary to maintain a hardware coherent global memory abstraction. By relying on existing consistency mechanisms, we expect these extensions to only add minimal hardware complexity. This will ease integration into existing systems.

Within this project we have investigated several different options for placing and integrating these extensions. The most promising was an integration into the coherence controller on the memory side of the system. At this location, it is easy to track memory translations and the extensions can be built on top of the existing coherence state machines that are used for the existing coherence support.

The problem of memory coherence for remapped memory regions, however, is only a subset of a more general problem description. Several copies of the same physical data at multiple locations can also be introduced by several other sources. Examples are reconfigurable caches that dynamically change the storage location for cached data and/or are capable of storing the data at two locations. Like the proposed remapping, this has the potential to significantly improve memory performance, but is also limited by the need to maintain coherence for the sake of easy programmability.

[3]TLB misses create a major memory bottleneck for many applications — especially for those with very large working sets and irregular access patterns.

The solution proposed in this project is therefore designed in a way that is generally capable of solving the coherence problem for arbitrary aliased memory regions, independent of the source for their aliasing. To accomplish that, the design is split into two orthogonal components: a) the coherence engine for aliased memory regions, and b) proposals on how to describe memory remapping and/or aliasing. While the former component is generally applicable, the second component is optimization-specific and independently specifies how aliased memory regions are introduced into the system and how they can be detected. Between these two components is a well defined interface, which allows the exchange of the alias engine without impacting the extended coherence controller. This leads to a general and highly flexible solution and enables the safe use of memory aliasing.

The initial design of the generalized coherence controller has been completed and has been formalized. Based on this formal model, the system is currently being implemented in a simulation framework for experimental testing. As for the second component, the aliasing of memory regions, we currently focus only on remapping for improved memory performance. This part is based on and will reuse software components from the Impulse project. Together, the overall system will show the benefits of both having a generalized coherence controller for aliased memory regions and the advantages of deploying memory remapping in shared memory systems without an impact on the programming model.

4.3 Simulation Environment

As with any large-scale architectural project, we use simulation to investigate the large design space. This simulation infrastructure, however, is currently still under development. We have investigated a large range of simulation tools and none of the available ones fits to the requirements in this project. In particular, the memory system models in existing simulators are not powerful enough to model the extensions envisioned in this project. This limitation leads us to design of our own memory backend on top of an existing processor core simulator. We will mainly leverage two existing simulators (due to prior experience): SIMT [7], a NUMA multiprocessor simulator based on Augmint [4], and URSIM [8], which is based on RSIM [5] and was developed within the Impulse project at the University of Utah [6].

5 Conclusion and Future Work

This project has explored novel techniques to improve memory system performance in SMP-based systems for applications important to LLNL. We have focused on a range of techniques to overcome the performance bottleneck of current multiprocessor systems and to increase single-node efficiency.

Our work has included the design and implementation of a toolset to analyze memory access patterns of applications, the exploration of regularity metrics and their use to classify code behavior, and a set of microbenchmarks to assess and quantify the performance of SMP memory systems. Portions of this work are the basis of a University of Utah Master's Thesis, and we are currently preparing a conference paper on them. We plan to improve support for threaded applications in the toolset, and will make these tools available to the general laboratory user community to help the evaluation and optimization of LLNL applications.

In addition, we have explored mechanisms to maintain memory coherence in SMP-based systems that use DAO techniques. We have identified that these mechanisms solve an aliasing problem that occurs with many memory system techniques, and have applied for NSF funding to continue this research.

References

[1] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[2] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79, January 1999.

[3] International Business Machines Inc. *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*, 1998.

[4] A-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The augmint multiprocessor simulation toolkit for intel x86 ar chitectures. In *Proceedings of 1996 International Conference on Computer Desig n*, October 1996.

[5] V.S. Pai, P. Ranganathan, and S.V. Adve. RSIM reference manual, version 1.0. *IEEE Technical Committee on Computer Architecture Newsletter*, Fall 1997.

[6] The Impulse Adaptable Memory System Project. Impulse: Building a smarter memory controller. http://www.cs.utah.edu/impulse, 2002.

[7] Jie Tao, Wolfgang Karl, and Martin Schulz. Using Simulation to Understand the Data Layout of Programs. In *Proceedings of the IASTED International Conference on Applied Simulation and Modelling (ASM 2001)*, pages 349–354, September 2001.

[8] Lixin Zhang. URSIM reference manual. Technical Report UUCS-00-015, University of Utah, August 2000.